(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 247 172 B1

(12) **EUROPEAN PATENT SPECIFICATION**

(45) Date of publication and mention of the grant of the patent:
29.09.2004 Bulletin 2004/40

(21) Application number: **00950458.0**

(22) Date of filing: **19.07.2000**

(51) Int Cl.[7]: **G06F 9/32**

(86) International application number:
PCT/US2000/019770

(87) International publication number:
WO 2001/052059 (19.07.2001 Gazette 2001/29)

(54) **CALL GATE EXPANSION FOR 64 BIT ADDRESSING**

ANRUFGATTEREXPANSION FÜR 64 BIT ADRESSIERUNG

EXTENSION DE LA PASSERELLE D'APPEL POUR ADRESSAGE 64 BITS

(84) Designated Contracting States:
DE GB

(30) Priority: **14.01.2000 US 483078**

(43) Date of publication of application:
09.10.2002 Bulletin 2002/41

(73) Proprietor: **ADVANCED MICRO DEVICES INC.**
Sunnyvale, California 94088-3453 (US)

(72) Inventor: **MCGRATH, Kevin, J.**
Los Gatos, CA 95033 (US)

(74) Representative: **Wright, Hugh Ronald**
Brookes Batchellor,
102-108 Clerkenwell Road
London EC1M 5SA (GB)

(56) References cited:
US-A- 4 620 274
US-A- 5 481 684
US-A- 5 774 686
US-A- 5 784 638

EP 1 247 172 B1

Description

Field of the Invention

[0001] This invention is related to the field of processors and, more particularly, to address and operand sizes in processors.

Description of the Related Art

[0002] The x86 architecture (also known as the IA-32 architecture) has enjoyed widespread acceptance and success in the marketplace. Accordingly, it is advantageous to design processors according to the x86 architecture. Such processors may benefit from the large body of software written to the x86 architecture (since such processors may execute the software and thus computer systems employing the processors may enjoy increased acceptance in the market due to the large amount of available software).

[0003] As computer systems have continued to evolve, 64 bit address size (and sometimes operand size) has become desirable. A larger address size allows for programs having a larger memory footprint (the amount of memory occupied by the instructions in the program and the data operated upon by the program) to operate within the memory space. A larger operand size allows for operating upon larger operands, or for more precision in operands. More powerful applications and/or operating systems may be possible using 64 bit address and/or operand sizes.

[0004] Unfortunately, the x86 architecture is limited to a maximum 32 bit operand size and 32 bit address size. The operand size refers to the number of bits operated upon by the processor (e.g. the number of bits in a source or destination operand). The address size refers to the number of bits in an address generated by the processor. Thus, processors employing the x86 architecture may not serve the needs of applications which may benefit from 64 bit address or operand sizes.

[0005] US-A-5 481 684 (PATTIN JAY C ET AL) 2 January 1996 (1996-01-02) discloses gate descriptors and segment descriptors in the x86 architecture. The gate descriptors control access to entry points into a code segment. They consist of two 4-byte double words and comprise an offset which determines the beginning address of the service routine within the segment. The segment descriptors comprise the beginning address of the segment, a segment limit and many attribute bits to control access to the segment and further define said segment. Said segment descriptors also consist of two 4-byte double words.

Summary of the Invention

[0006] The problems outlined above are in large part solved by a processor as described herein. The processor supports a first processing mode in which the address size is greater than 32 bits. The address size may be nominally indicated as 64 bits, although various embodiments of the processor may implement any address size which exceeds 32 bits, up to and including 64 bits, in the first processing mode. The first processing mode may be established by placing an enable indication in a control register into an enabled state and by setting a first operating mode indication and a second operating mode indication in a segment descriptor to predefined states. Other combinations of the first operating mode indication and the second operating mode indication may be used to provide compatibility modes for 32 bit and 16 bit processing compatible with the x86 processor architecture (with the enable indication remaining in the enabled state).

[0007] While the compatibility modes may allow 32 bit or 16 bit code to execute while the first processing mode is enabled via the enable indication, it may be desirable to call code operating in the first processing mode from the 32 bit or 16 bit code. For example, the operating system may operate in the first processing mode while application programs may operate in 32 or 16 bit mode. A call gate descriptor is defined which occupies two entries in a segment descriptor table. By occupying two entries, each of which may otherwise store a segment descriptor, the call gate descriptor may include enough space to store an address in excess of 32 bits. Thus, a calling code segment may reference a call gate descriptor, which may reference the target code segment and may provide an address within the address space of the target code segment, even if the address exceeds the address size in the calling code segment. Furthermore, by having the call gate descriptor occupy two entries, the segment descriptor table may continue to store segment descriptors for compatibility mode segments. Thus, call gate descriptors and compatibility mode segment descriptors may coexist in the segment descriptor table. Additionally, the area which would be the type field in the second entry occupied by the call gate descriptor may be coded to an invalid type, so that an inadvertent use of the second entry for a code segment may result in the processor signalling an exception.

[0008] Broadly speaking, a processor is contemplated. The processor comprises an execution core configured to execute a branch instruction specifying a segment selector. The processor is configured to read at least a first entry from a segment descriptor table responsive to the segment selector, and, if the first entry indicates a call gate descriptor, a second entry in the segment descriptor table stores a remaining portion of the call gate descriptor. Additionally, a computer system is contemplated comprising the processor and an input/output (I/O) device configured to communicate between the computer system and another computer system to which the I/O device is couplable.

[0009] Moreover, A method is contemplated. A call gate descriptor is read from a segment descriptor table. The call gate descriptor comprises a first entry and a

second entry in the segment descriptor table, wherein each of the first entry and the second entry is capable of storing a segment descriptor. An offset is extracted from the call gate descriptor. The offset locates a first instruction to be executed in a target code segment.

Brief Description of the Drawings

[0010] Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

Fig. 1 is a block diagram of one embodiment of a processor.

Fig. 2 is a block diagram of one embodiment of a segment descriptor for 32/64 mode.

Fig. 3 is a block diagram of one embodiment of a segment descriptor for compatibility mode.

Fig. 4 is a block diagram of operation in compatibility mode and in legacy mode according to one embodiment of the processor shown in Fig. 1.

Fig. 5 is a table illustrating one embodiment of operating modes as a function of segment descriptor and control register values.

Fig. 6 is a table illustrating one embodiment of the use of instruction prefixes to override default operating modes.

Fig. 7 is a block diagram of one embodiment of a register.

Fig. 8 is a diagram illustrating one embodiment of a global descriptor table and a local descriptor table.

Fig. 9 is a block diagram of one embodiment of a 32/64 call gate descriptor.

Fig. 10 is a block diagram of an instruction format.

Fig. 11 is a block diagram of one embodiment of a computer system including the processor shown in Fig. 1.

Fig. 12 is a block diagram of another embodiment of a computer system including the processor shown in Fig. 1.

[0011] While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

Detailed Description of the Preferred Embodiments

[0012] Turning now to Fig. 1, a block diagram illustrating one embodiment of a processor 10 is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 1, processor 10 includes an instruction cache 12, an execution core 14, a data cache 16, an external interface unit 18, a memory management unit (MMU) 20, and a register file 22. In the illustrated embodiment, MMU 20 includes a set of segment registers 24, a first control register 26, a second control register 28, a local descriptor table register (LDTR) 30, and a global descriptor table register (GDTR) 32. Instruction cache 12 is coupled to external interface unit 18, execution core 14, and MMU 20. Execution core 14 is further coupled to MMU 20, register file 22, and data cache 16. Data cache 16 is further coupled to MMU 20 and external interface unit 18. External interface unit 18 is further coupled to MMU 20 and to an external interface.

[0013] Generally speaking, processor 10 employs a processor architecture compatible with the x86 architecture and including additional architectural features to support 64 bit processing. Processor 10 is configured to establish an operating mode in response to information stored in a code segment descriptor corresponding to the currently executing code and further in response to one or more enable indications stored in one or more control registers. As used herein, an "operating mode" specifies default values for various programmably selectable processor attributes. For example, the operating mode may specify a default operand size and a default address size. The default operand size specifies the number of bits in an operand of an instruction, unless an instruction's encoding overrides the default. The default address size specifies the number of bits in an address of a memory operand of an instruction, unless an instruction's encoding overrides the default. The default address size specifies the size of at least the virtual address of memory operands, and may also specify the size of the physical address. Alternatively, the size of the physical address may be independent of the default address size and may instead be dependent on the LME bit described below (e.g. the physical address may be 32 bits if the LME bit is clear and an implementation-dependent size greater than 32 bits and less than 64 bits if the LME bit is set) or on another control bit (e.g. the physical address extension bit, or PAE bit, in another control register). As used herein, a "virtual address" is an address generated prior to translation through an address translation mechanism (e.g. a paging mechanism) to a "physical address", which is the address actually used to access a memory. Additionally, as used

herein, a "segment descriptor" is a data structure created by software and used by the processor to define access control and status for a segment of memory. A "segment descriptor table" is a table in memory having multiple entries, each entry capable of storing a segment descriptor.

[0014] In the illustrated embodiment, MMU 20 generates an operating mode and conveys the operating mode to execution core 14. Execution core 14 executes instructions using the operating mode. More particularly, execution core 14 fetches operands having the default operand size from register file 22 or memory (through data cache 16, if the memory operands are cacheable and hit therein, or through external interface unit 18 if the memory operands are noncacheable or miss data cache 16) unless a particular instruction's encoding overrides the default operand size, in which case the overriding operand size is used. Similarly, execution core 14 generates addresses of memory operands, wherein the addresses have the default address size unless a particular instruction's encoding overrides the default address size, in which case the overriding address size is used. In other embodiments, the information used to generate the operating mode may be shadowed locally in the portions of processor 10 which use the operating mode (e.g. execution core 14), and the operating mode may be determined from the local shadow copies.

[0015] As mentioned above, MMU 20 generates the operating mode responsive to a code segment descriptor corresponding to the code being executed and further responsive to one or more values in control registers. Information from the code segment descriptor is stored in one of the segment registers 24 (a register referred to as CS, or code segment). Additionally, control register 26 stores an enable indication (LME) which is used to enable an operating mode in which the default address size is greater than 32 bits ("32/64 mode") as well as certain compatibility modes for the 32 bit and 16 bit operating modes. The default operand size may be 32 bits in 32/64 mode, but instructions may override the default 32 bit operand size with a 64 bit operand size when desired. If the LME indication is in an enabled state, then 32/64 mode may be used in addition to 32 bit and 16 bit modes. If the LME indication is in a disabled state, then 32/64 mode is disabled. In one embodiment, the default address size in 32/64 mode may be implementation-dependent but may be any value up to and including 64 bits. Furthermore, the size of the virtual address may differ in a given implementation from the size of the physical address in that implementation.

[0016] It is noted that enable indications may be described herein as bits with the enabled state being the set state of the bit and the disabled state being the cleared state of the bit. However, other encodings are possible, including encodings in which multiple bits are used and encodings in which the enabled state is the clear state and the disabled state is the set state. Accordingly, the remainder of this description may refer to the LME indication in control register 26 as the LME bit, with the enabled state being set and the disabled state being clear. However, other encodings of the LME indication are contemplated, as set forth above.

[0017] Segment registers 24 store information from the segment descriptors currently being used by the code being executed by processor 10. As mentioned above, CS is one of segment registers 24 and specifies the code segment of memory. The code segment stores the code being executed. Other segment registers may define various data segments (e.g. a stack data segment defined by the SS segment register, and up to four data segments defined by the DS, ES, FS, and GS segment registers). Fig. 1 illustrates the contents of an exemplary segment register 24A, including a selector field 24AA and a descriptor field 24AB. Selector field 24AA is loaded with a segment selector to activate a particular segment in response to certain segment load instructions executed by execution core 14. The segment selector identifies the segment descriptor in a segment descriptor table in memory. More particularly, processor 10 may employ two segment descriptor tables: a local descriptor table and a global descriptor table. The base address of the local descriptor table is stored in the LDTR 30. Similarly, the base address of the global descriptor table is stored in GDTR 32. A bit within the segment selector (the table indicator bit) selects the descriptor table, and the remainder of the segment selector is used as an index into the selected table. When an instruction loads a segment selector into one of segment registers 24, MMU 20 reads the corresponding segment descriptor from the selected segment descriptor table and stores information from the segment descriptor into the segment descriptor field (e.g. segment descriptor field 24AB for segment register 24A). The information stored in the segment descriptor field may comprise any suitable subset of the segment descriptor, including all of the segment descriptor, if desired. Additionally, other information derived from the segment descriptor or other sources may be stored in the segment descriptor field, if desired. For example, an embodiment may decode the operating mode indications from the code segment descriptor and store the decoded value rather than the original values of the operating mode indications. If an instruction causes CS to be loaded with a segment selector, the code segment may change and thus the operating mode of processor 10 may change. Segment descriptor tables are described in more detail below.

[0018] In one embodiment, only the CS segment register is used in 32/64 mode. The data segment registers are ignored. In 16 and 32 bit modes, the code segment and data segments may be active. Furthermore, a second enable indication (PE) in control register 28 may affect the operation of MMU 20. The PE enable indication may be used to enable protected mode, in which segmentation and/or paging address translation mechanisms may be used. If the PE enable indication is in the

disabled state, segmentation and paging mechanisms are disabled and processor 10 is in "real mode" (in which addresses generated by execution core 14 are physical addresses). Similar to the LME indication, the PE indication may be a bit in which the enabled state is the bit being set and the disabled state is the bit being clear. However, other embodiments are contemplated as described above.

[0019] It is noted that MMU 20 may employ additional hardware mechanisms, as desired. For example, MMU 20 may include paging hardware to implement paging address translation from virtual addresses to physical addresses. The paging hardware may include a translation lookaside buffer (TLB) to store page translations.

[0020] It is noted that control registers 26 and 28 may be implemented as architected control registers (e.g. control register 26 may be CR4 and control register 28 may be CR0). Alternatively, one or both of the control registers may be implemented as model specific registers to allow for other uses of the architected control registers without interfering with 32/64 mode.

[0021] Generally, instruction cache 12 is a high speed cache memory for storing instruction bytes. Execution core 14 fetches instructions from instruction cache 12 for execution. Instruction cache 12 may employ any suitable cache organization, including direct-mapped, set associative, and fully associative configurations. If an instruction fetch misses in instruction cache 12, instruction cache 12 may communicate with external interface unit 18 to fill the missing cache line into instruction cache 12. Additionally, instruction cache 12 may communicate with MMU 20 to receive physical address translations for virtual addresses fetched from instruction cache 12.

[0022] Execution core 14 executes the instructions fetched from instruction cache 12. Execution core 14 fetches register operands from register file 22 and updates destination registers in register file 22. The size of the register operands is controlled by the operating mode and any overrides of the operating mode for a particular instruction. Similarly, execution core 14 fetches memory operands from data cache 16 and updates destination memory locations in data cache 16, subject to the cacheability of the memory operands and hitting in data cache 16. The size of the memory operands is similarly controlled by the operating mode and any overrides of the operating mode for a particular instruction. Furthermore, the size of the addresses of the memory operands generated by execution core 14 is controlled by the operating mode and any overrides of the operating mode for a particular instruction.

[0023] Execution core 14 may employ any suitable construction. For example, execution core 14 may be a superpipelined core, a superscalar core, or a combination thereof. Execution core 14 may employ out of order speculative execution or in order execution, according to design choice.

[0024] Register file 22 may include 64 bit registers which may be accessed as 64 bit, 32 bit, 16 bit, or 8 bit registers as indicated by the operating mode of processor 10 and any overrides for a particular instruction. The register format for one embodiment is described below with respect to Fig. 7. The registers included in register file 22 may include the LEAX, LEBX, LECX, LEDX, LEDI, LESI, LESP, and LEBP registers. Register file 22 may further include the LEIP register. Alternatively, execution core 14 may employ a form of register renaming in which any register within register file 22 may be mapped to an architected register. The number of registers in register file 22 may be implementation dependent for such an embodiment.

[0025] Data cache 16 is a high speed cache memory configured to store data. Data cache 16 may employ any suitable cache organization, including direct-mapped, set associative, and fully associative configurations. If a data fetch or update misses in data cache 16, data cache 16 may communicate with external interface unit 18 to fill the missing cache line into data cache 16. Additionally, if data cache 16 employs a writeback caching policy, updated cache lines which are being cast out of data cache 16 may be communicated to external interface unit 18 to be written back to memory. Data cache 16 may communicate with MMU 20 to receive physical address translations for virtual addresses presented to data cache 16.

[0026] External interface unit 18 communicates with portions of the system external to processor 10. External interface unit 18 may communicate cache lines for instruction cache 12 and data cache 16 as described above, and may communicate with MMU 20 as well. For example, external interface unit 18 may access the segment descriptor tables and/or paging tables on behalf of MMU 20.

[0027] It is noted that processor 10 may include an integrated level 2 (L2) cache, if desired. Furthermore, external interface unit 18 may be configured to communicate with a backside cache in addition to communicating with the system.

[0028] Turning now to Fig. 2, a block diagram of one embodiment of a code segment descriptor 40 for 32/64 mode is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 2, code segment descriptor 40 comprises 8 bytes with the most significant 4 bytes illustrated above the least significant 4 bytes. The most significant four bytes are stored at a numerically larger address than the least significant four bytes. The most significant bit of each group of four bytes is illustrated as bit 31 in Fig. 2 (and Fig. 3 below), and the least significant bit is illustrated as bit 0. Short vertical lines within the four bytes delimit each bit, and the long vertical lines delimit a bit but also delimit a field (both in Fig. 2 and in Fig. 3).

[0029] Unlike the 32 bit and 16 bit code segment descriptors illustrated in Fig. 3 below, code segment descriptor 40 does not include a base address or limit. Processor 10 employs a flat virtual address space for 32/64 mode (rather than the segmented linear address

space employed in 32 bit and 16 bit modes). Accordingly, the portions of code segment descriptor 40 which would otherwise store the base address and limit are reserved in segment descriptor 40. It is noted that a virtual address provided through segmentation may also be referred to herein as a "linear address". The term "virtual address" encompasses any address which is translated through a translation mechanism to a physical address actually used to address memory, including linear addresses and other virtual addresses generated in non-segmented architectures.

[0030] Segment descriptor 40 includes a D bit 42, an L bit 44 (set to one for a 32/64 mode code segment), an available bit (AVL) 46, a present (P) bit 48, a descriptor privilege level (DPL) 50, and a type field 52. D bit 42 and L bit 44 are used to determine the operating mode of processor 10, as illustrated in Fig. 5 below. AVL bit 46 is available for use by system software (e.g. the operating system). P bit 48 is used to indicate whether or not the segment is present in memory. If P bit 48 is set, the segment is present and code may be fetched from the segment. If P bit 48 is clear, the segment is not present and an exception is generated to load the segment into memory (e.g. from disk storage or through a network connection). The DPL indicates the privilege level of the segment. Processor 10 employs four privilege levels (encoded as 0 through 3 in the DPL field, with level 0 being the most privileged level). Certain instructions and processor resources (e.g. configuration and control registers) are only executable or accessible at the more privileged levels, and attempts to execute these instructions or access these resources at the lower privilege levels result in an exception. When information from code segment 40 is loaded into the CS segment register, the DPL becomes the current privilege level (CPL) of processor 10. Type field 52 encodes the type of segment. For code segments, the most significant bit two bits of type field 52 may be set (the most significant bit distinguishing a code or data segment from a system segment, and the second most significant bit distinguishing a code segment from a data segment), and the remaining bits may encode additional segment type information (e.g. execute only, execute and read, or execute and read only, conforming, and whether or not the code segment has been accessed).

[0031] It is noted that, while several indications in the code segment descriptor are described as bits, with set and clear values having defined meanings, other embodiments may employ the opposite encodings and may use multiple bits, as desired. Thus, for example, the D bit 42 and the L bit 44 may each be an example of an operating mode indication which may be one or more bits as desired, similar to the discussion of enable indications above.

[0032] Turning now to Fig. 3, a block diagram of one embodiment of a code segment descriptor 54 for 32 and 16 bit compatibility mode is shown. Other embodiments are possible and contemplated. As with the embodiment of Fig. 2, code segment descriptor 54 comprises 8 bytes with the most significant 4 bytes illustrated above the least significant 4 bytes.

[0033] Code segment descriptor 54 includes D bit 42, L bit 44, AVL bit 46, P bit 48, DPL 50, and type field 52 similar to the above description of code segment descriptor 40. Additionally, code segment descriptor 54 includes a base address field (reference numerals 56A, 56B, and 56C), a limit field (reference numerals 57A and 57B) and a G bit 58. The base address field stores a base address which is added to the logical fetch address (stored in the LEIP register) to form the linear address of an instruction, which may then optionally be translated to a physical address through a paging translation mechanism. The limit field stores a segment limit which defines the size of the segment. Attempts to access a byte at a logical address greater than the segment limit are disallowed and cause an exception. G bit 58 determines the scaling of the segment limit field. If G bit 58 is set the limit is scaled to 4K byte pages (e.g. 12 least significant zeros are appended to the limit in the limit field). If G bit 58 is clear, the limit is used as is.

[0034] It is noted that code segment descriptors for 32 and 16 bit modes when 32/64 mode is not enabled via the LME bit in control register 26 may be similar to code segment descriptor 54, except the L bit is reserved and defined to be zero. It is further noted that, in 32 and 16 bit modes (both compatibility mode with the LME bit set and modes with the LME bit clear) according to one embodiment, data segments are used as well. Data segment descriptors may be similar to code segment descriptor 54, except that the D bit 42 is defined to indicate the upper bound of the segment or to define the default stack size (for stack segments).

[0035] Tuming next to Fig. 4, a diagram illustrating exemplary uses of the LME bit in control register 26 and the compatibility modes to allow for a high degree of flexibility in implementing the 32/64 mode and the 32 and 16 bit modes is shown. A box 60 illustrates exemplary operation when the LME bit is set, and a box 62 illustrates exemplary operation when the LME bit is clear.

[0036] As illustrated in box 60, the compatibility modes supported when the LME bit is set may allow for a 64 bit operating system (i.e. an operating system designed to take advantage of the virtual and physical address spaces in excess of 32 bits and/or data operands of 64 bits) to operate with a 32 bit application program (i.e. an application program written using 32 bit operand and address sizes). The code segment for the operating system may be defined by the 32/64 mode code segment descriptor 40 illustrated in Fig. 2, and thus the L bit may be set. Accordingly, the operating system may take advantage of the expanded virtual address space and physical address space for the operating system code and the data structures maintained by the operating system (including, e.g. the segment descriptor tables and the paging translation tables). The operating system may also use the 64 bit data type defined in

32/64 mode using instruction encodings which override the default 32 bit operand size. Furthermore, the operating system may launch a 32 bit application program by establishing one or more 32 bit compatibility mode segment descriptors (L bit cleared, D bit set, e.g. segment descriptor 54 shown in Fig. 2) in the segment descriptor table and branching into one of the compatibility mode segments. Similarly, the operating system may launch a 16 bit application program by establishing one or more 16 bit compatibility mode segment descriptors (L bit cleared, D bit cleared, e.g. segment descriptor 54 shown in Fig. 2) in the segment descriptor table and branching into one of the compatibility mode segments. Accordingly, a 64 bit operating system may retain the ability to execute existing 32 bit and 16 bit application programs in the compatibility mode. A particular application program may be ported to 32/64 mode if the expanded capabilities are desired for that program, or may remain 32 bit or 16 bit.

[0037] While processor 10 is executing the 32 bit application program, the operating mode of processor 10 is 32 bit. Thus, the application program may generally execute in the same fashion as it does in 32 bit mode with the LME bit clear (e.g. when the operating system is a 32 bit operating system as well). However, the application program may call an operating system service, experience an exception, or terminate. In each of these cases, processor 10 may return to executing operating system code (as illustrated by arrow 64 in Fig. 4). Since the operating system code operates in 32/64 mode, the address of the operating system service routine, exception handler, etc. may exceed 32 bits. Thus, processor 10 may need to generate an address greater than 32 bits prior to returning to the operating system code. The LME bit provides processor 10 with an indication that the operating system may be operating in 32/64 mode even though the current operating mode is 32 bit, and thus processor 10 may provide the larger address space for operating system calls and exceptions.

[0038] In one embodiment, exceptions are handled using interrupt segment descriptors stored in an interrupt segment descriptor table. If the LME bit is set, the interrupt segment descriptors may be 16 byte entries which include a 64 bit address of the operating system routine which handles the exception. If the LME bit is clear, the interrupt segment descriptors may be eight byte entries which include a 32 bit address. Accordingly, processor 10 accesses the interrupt descriptor table responsive to the LME indication (i.e. reading a 16 byte entry if the LME bit is set and reading an eight byte entry if the LME bit is clear). Therefore, exceptions may be handled by the 64 bit operating system even though the application program is executing in 32 bit compatibility mode. Furthermore, processor 10 supports a 32 bit (or 16 bit) operating system if the LME bit is clear.

[0039] Similarly, the call mechanisms within processor 10 may operate in different fashions based on the state of the LME bit. Since the operating system typically executes at a higher privilege level than the application program, transfers from the application program to the operating system are carefully controlled to ensure that the application program is only able to execute permitted operating system routines. More generally, changes in privilege level are carefully controlled. In one embodiment, processor 10 may support at least two mechanisms for performing operating system calls. One method may be through a call gate in the segment descriptor tables (described in more detail below). Another method may be the SYSCALL instruction supported by processor 10, which uses a model specific register as the source of the address of the operating system routine. Updating the model specific registers is a privileged operation, and thus only code executing at a higher privilege level (e.g. operating system code) may establish the address in the model specific register used by the SYSCALL instruction. For the SYSCALL method, a second model specific register may be defined to store the most significant 32 bits of the address of the operating system routine. Thus, if the LME bit is set, the address may be read from the two model specific registers. If the LME bit is clear, the address may be read from the model specific register storing the least significant 32 bits. Alternatively, the model specific register used by the SYSCALL instruction may be expanded to 64 bits and the address may be 32 bits (the least significant 32 bits of the model specific register) or 64 bits based on the state of the LME bit.

[0040] As illustrated above, having the LME bit set may allow for processor 10 to operate in a system in which the operating system is 64 bit and one or more application programs are not 64 bit (e.g. 32 bit as shown or 16 bit, which operates in a similar fashion to the above description). Additionally, as illustrated by box 62, having the LME bit clear may allow for processor 10 to operate in 32 bit or 16 bit modes compatible with the x86 architecture. As described above, the mechanisms for handling exceptions and operating system calls are designed to handle the LME bit being set or clear, and thus the 32 bit and 16 bit modes may operate unmodified, even though processor 10 is capable of operating in 32/64 mode. Furthermore, by providing the x86 compatible 16 and 32 bit modes when the LME bit is clear, (and ignoring the L bit, which is reserved in these modes) processor 10 may operate in a system in which the L bit is defined for some other purpose than for 32/64 mode and may still support 32/64 mode if the LME bit is set. Accordingly, a system employing a 32 bit operating system and 32 bit or 16 bit application programs may employ processor 10. Subsequently, the system could be upgraded to a 64 bit operating system without having to change processor 10.

[0041] Not illustrated in Fig. 4 is a 64 bit operating system and a 64 bit application program operating with the LME bit set. The mechanisms for calling operating system routines described above for the 64 bit operating system and 32 bit application program may apply equal-

ly to the 64 bit application program as well. Additionally, call gates which support 64 bits of offset are supported (as will be described in more detail below).

[0042] Turning next to Fig. 5, a table 70 is shown illustrating the states of the LME bit, the L bit in the code segment descriptor, and the D bit in the code segment descriptor and the corresponding operating mode of processor 10 according to one embodiment of processor 10. Other embodiments are possible and contemplated. As table 70 illustrates, if the LME bit is clear, then the L bit is reserved (and defined to be zero). However, processor 10 may treat the L bit as a don't care if the LME bit is clear. Thus, the x86 compatible 16 bit and 32 bit modes may be provided by processor 10 if the LME bit is clear. If the LME bit is set and the L bit in the code segment is clear, then a compatibility operating mode is established by processor 10 and the D bit selects 16 bit or 32 bit mode. If the LME bit and the L bit are set and the D bit is clear, 32/64 mode is selected for processor 10. Finally, the mode which would be selected if the LME, L and D bits are all set is reserved.

[0043] As mentioned above and illustrated in Fig. 6 below, the 32/64 operating mode includes a default address size in excess of 32 bits (implementation dependent but up to 64 bits) and a default operand size of 32 bits. The default operand size of 32 bits may be overridden to 64 bits via a particular instruction's encoding. The default operand size of 32 bits is selected to minimize average instruction length (since overriding to 64 bits involves including an instruction prefix in the instruction encoding which may increase the instruction length) for programs in which 32 bits are sufficient for many of the data manipulations performed by the program. For such programs (which may be a substantial number of the programs currently in existence), moving to a 64 bit operand size may actually reduce the execution performance achieved by the program (i.e. increased execution time). In part, this reduction may be attributable to the doubling in size in memory of the data structures used by the program when 64 bit values are stored. If 32 bits is sufficient, these data structures would store 32 bit values, Thus, the number of bytes accessed when the data structure is accessed increases if 64 bit values are used where 32 bit values would be sufficient, and the increased memory bandwidth (and increased cache space occupied by each value) may cause increased execution time. Accordingly, 32 bits is selected as the default operand size and the default may be overridden via the encoding of a particular instruction.

[0044] Turning next to Fig. 6, a table 72 is shown illustrating one embodiment of the use of instruction prefixes to override the operating mode for a particular instruction. Other embodiments are possible and contemplated. Execution core 14 determines the address size and operand size for a particular instruction according to table 72. In particular for the embodiment illustrated in Fig. 6, an instruction prefix byte (the address size override prefix byte) may be used to override the default address size and another instruction prefix byte (the operand size override prefix byte) may be used to override the default operand size. The address size override prefix byte is encoded as 67 (in hexadecimal) and the operand size override prefix byte is encoded as 66 (in hexadecimal). The number of override prefixes in the particular instruction forms the columns of the table. The rows of the table indicate the operand size and address size of the particular instruction, based on the operating mode and the number of override prefixes in the corresponding column. The number of override prefixes refers to the number of override prefixes of the corresponding type (e.g. address size rows are the address size based on the number of address size override prefixes and operand size rows are the operand size based on the number of operand size override prefixes).

[0045] The column labeled "0" for the number of override prefixes illustrates the default operand size and address size for each operating mode. It is noted that the 32 bit and 16 bit mode rows refer to both the compatibility modes (LME set) and the standard modes (LME clear). Furthermore, while the default address size is 64 bits in 32/64 mode, the actual number of address bits may be implementation dependent, as discussed above.

[0046] The inclusion of one address size override prefix in 32/64 bit mode changes the address size from 64 bit (which may be less than 64 bits for a given implementation but is greater than 32 bits) to 32 bit, as shown in table 72. Additionally, the inclusion of one operand size override prefix in 32/64 bit mode changes the operand size from 32 bit to 64 bit. It may be desirable to provide for a 16 bit operand as well (e.g. to support the short integer data type in the "C" programming language). Accordingly, the inclusion of two operand size override prefixes in 32/64 mode selects an operand size of 16 bits. The inclusion of more than two operand size override prefixes results in the same operand size as the inclusion of two operand size override prefixes. Similarly, the inclusion of more than one address size override prefix results in the same address size as the inclusion of one address size override prefix.

[0047] For the 32 bit modes, the inclusion of one override prefix toggles the default 32 bit size to 16 bit, and the inclusion of more than one override prefix has the same effect as the inclusion of one override prefix. Similarly, for 16 bit modes, the inclusion of one override prefix toggles the default 16 bit size to 32 bit, and the inclusion of more than one override prefix has the same effect as the inclusion of one override prefix.

[0048] Turning now to Fig. 7, a diagram illustrating one embodiment of the LEAX register 74 is shown. Other registers within register file 22 may be similar. Other embodiments are possible and contemplated. In the embodiment of Fig. 7, register 74 includes 64 bits, with the most significant bit labeled as bit 63 and the least significant bit labeled as bit 0. Fig. 7 illustrates the portions of the LEAX register accessed based upon the op-

erand size of an instruction (if the A register is selected as an operand). More particularly, the entirety of register 74 is accessed if the operand size is 64 bits (as illustrated by the brace labeled "LEAX" in Fig. 7). If the operand size is 32 bits, bits 31:0 of register 74 are accessed (as illustrated by the brace labeled "EAX" in Fig. 7). If the operand size is 16 bits, bits 16:0 of the register are accessed (as illustrated by the brace labeled "AX" in Fig. 7). The above operand sizes may be selected based on the operating mode and the inclusion of any override prefixes. However, certain instruction opcodes are defined which access an eight bit register (AH or AL in Fig. 7).

[0049] Turning next to Fig. 8, a block diagram is shown illustrating one embodiment of a global descriptor table 80 and a local descriptor table 82. Other embodiments are possible and contemplated. As illustrated in Fig. 8 and mentioned above, the base address of global descriptor table 80 is provided by GDTR 32 and the base address of local descriptor table 82 is provided by LDTR 30. Accordingly, to support placing global descriptor table 80 and local descriptor table 82 arbitrarily within the virtual address space, GDTR 32 and LDTR 30 may store 64 bit base addresses. If the LME bit is clear, the least significant 32 bits of the base address may be used to locate the descriptor tables.

[0050] Both global descriptor table 80 and local descriptor table 82 are configured to store segment descriptors of various types. For example, 32/64 mode code segment descriptors 84, 86, and 90 and compatibility mode descriptors 92 and 94 are illustrated in Fig. 8. Each of descriptors 84-94 occupies an entry in the corresponding descriptor table, where an entry is capable of storing one segment descriptor (e.g. 8 bytes for the embodiments illustrated in Figs. 2 and 3). Another type of descriptor in global descriptor table 80 is a local descriptor table descriptor 96, which defines a system segment for the local descriptor table 82 and provides the base address stored in LDTR 30. LDTR 30 is initialized using an LLDT instruction having as an operand a segment selector locating descriptor 96 in global descriptor table 80. Global descriptor table 80 may store multiple LDT descriptors locating different local descriptor tables, if desired. Since the LDT descriptor 96 may store a 64 bit offset if the LME bit is set, LDT descriptor 96 may occupy two entries in global descriptor table 80. If the LME bit is clear, LDT descriptor 96 may occupy a single entry in global descriptor table 80. Similarly, each task may have a task state segment (TSS) descriptor in one of descriptor tables 80 and 82 to store certain information related to the task. Accordingly, a TSS descriptor may occupy two entries to allow for TSS information to be stored anywhere in the 64 bit address space.

[0051] The local and global descriptor tables may also store a call gate descriptor. For example, Fig. 8 illustrates call gate descriptors 100, 102, and 104. Call gate descriptors support a 64 bit offset as well, and thus may occupy two entries in the corresponding descriptor table as well. An exemplary 32/64 call gate descriptor is illustrated in Fig. 9 below.

[0052] By maintaining the segment descriptor tables 80 and 82 at 8 bytes and using two entries for descriptors which include 64 bit offsets, descriptors for 16 and 32 bit modes may be stored in the same tables as the descriptors which include 64 bit offsets. Thus, applications operating in compatibility modes may have appropriate descriptors in the same segment descriptor tables as the 64 bit operating systems.

[0053] Generally, call gates are used to manage the transition between a code segment having a lesser privilege level and a code segment have a greater privilege level (e.g. an application program calling an operating system routine). The lesser privileged code includes a call or other branch instruction specifying, as a target, a segment selector (and an offset into the segment, which is ignored in this case). The segment selector identifies a call gate descriptor within the descriptor tables, which includes a minimum privilege level required to execute the greater privilege level code. When processor 10 executes the call or other branch instruction, processor 10 indexes the descriptor tables with the segment selector and locates the call gate. If the current privilege level of processor 10 and the requestor privilege level (which is part of the segment selector, and may be used to lower the current privilege level for privilege checking purposes) both reflect sufficient privilege (e.g. the privilege levels are numerically less than or equal to the minimum privilege level in the call gate descriptor), then the call may proceed. The call gate descriptor includes a segment selector for the target segment (the code segment having the greater privilege level) and the offset within the target segment at which code fetching is to begin. Processor 10 extracts the segment selector and the offset from the call gate descriptor and reads the target segment descriptor to begin fetching the code having the greater privilege level. On the other hand, if either the current privilege level or the requestor privilege level is a lesser privilege level than the minimum privilege level in the call gate descriptor (e.g. either the current or requestor privilege level is numerically greater than the minimum privilege level), processor 10 signals an exception after accessing the call gate descriptor and without accessing the target descriptor. Thus, access to code executing at greater privilege levels is carefully controlled.

[0054] As mentioned above, the call gate descriptor includes a target segment selector and offset within the segment. The reference to the target segment descriptor is illustrated in Fig. 8 as an arrow from a call gate descriptor to another descriptor. For example, call gate descriptor 100 references mode descriptor 90; call gate descriptor 102 references 32/64 mode descriptor 86, and call gate descriptor 104 references 32/64 mode descriptor 84. As Fig. 8 illustrates, a call gate descriptor may be stored in either descriptor table and may reference a descriptor in the other table or in the same table.

Furthermore, a call gate descriptor may reference either a 32/64 mode descriptor or a compatibility mode descriptor.

[0055] Generally, when processor 10 reads a descriptor from one of the descriptor tables using a segment selector, one descriptor table entry is read. However, if the LME bit is set and processor 10 detects that the entry is a call gate descriptor, an LDT descriptor, or a TSS descriptor, processor 10 reads the next succeeding entry in the table to obtain the remainder of the descriptor. Accordingly, call gate descriptors, LDT descriptors, and TSS descriptors may coexist in a table with compatibility mode descriptors (or standard mode descriptors) which are of a different size, without redefining the size of the table entries nor how the table is managed for descriptors which occupy one entry. Furthermore, since the second portion of the call gate descriptor, the LDT descriptor, and the TSS descriptor may be accessed as a segment descriptor, the portion of the descriptor which would be the type field of a descriptor in the second portion is set to an invalid type when the descriptor is stored into the descriptor table, as shown below in Fig. 9. Alternatively, processor 10 may read two consecutive entries from a descriptor table each time a descriptor table read is performed, and the second entry may be used if the first entry is a call gate, LDT descriptor type, or TSS descriptor type.

[0056] It is noted that code operating in any operating mode (32/64 mode, 32 bit compatibility mode, or 16 bit compatibility mode) may reference a call gate descriptor when the LME bit is set. Thus, a 32 or 16 bit application may call an operating system routine even if the address of the routine is outside the 32 bit or 16 bit address space using the call gate mechanism. Additionally, a call gate descriptor may reference a code segment having any operating mode. The operating system may ensure that the most significant 32 bits of the offset in the call gate are zero (for a 32 bit target segment) or the most significant 48 bits of the offset in the call gate are zero (for a 16 bit target segment).

[0057] Turning now to Fig. 9, a block diagram of one embodiment of a call gate descriptor 120 is shown. Other embodiments are possible and contemplated. Similar to Figs. 2 and 3, the most significant bytes are illustrated above the least significant bytes. The most significant bit of each group of four bytes is illustrated as bit 31 and the least significant bit is illustrated as bit 0. Short vertical lines within the four bytes delimit each bit, and the long vertical lines delimit a bit but also delimit a field. As mentioned above, a call gate descriptor occupies two entries in a descriptor table. The horizontal dashed line in Fig. 9 divides call gate descriptor 120 into an upper portion (above the line) and a lower portion (below the line). The lower portion is stored in the entry indexed by the call gate's segment selector, and the upper portion is stored in the next succeeding entry.

[0058] Call gate descriptor 120 includes a target segment selector (field 122), an offset (fields 124A, 124B, and 124C), a present (P) bit 126, a descriptor privilege level (DPL). 128, a type field 130, and a pseudo-type field 132. The P bit is similar to P bit 48 described above. The target segment selector identifies an entry within one of the descriptor tables at which the target segment descriptor (having the greater privilege level) is stored. The offset identifies the address at which code fetching is to begin. In 32/64 mode, since the code segment has no base address and flat linear addressing is used, the offset is the address at which code fetching begins. In other modes, the offset is added to the segment base defined by the target segment descriptor to generate the address at which code fetching begins. As mentioned above, the offset may comprise 64 bits in the present embodiment.

[0059] DPL 128 stores the minimum privilege level of the calling routine must have (both in the current privilege level and the requested privilege level) which may successfully pass through the call gate and execute the called routine at the privilege level specified in the target segment descriptor.

[0060] Type field 130 is coded to a call gate descriptor type. In one embodiment, this type is coded as the 32 bit call gate type defined in the x86 architecture. Alternatively, other encodings may be used. Finally, pseudo-type field 132 is coded to an invalid type (e.g. zero) to ensure that if a segment selector identifying the segment table entry storing the upper half of call gate descriptor 120 is presented, then an exception will be signalled by processor 10.

[0061] It is noted that the lower half of LDT descriptor 96 may be similar to the 32 bit LDT descriptor and the upper half of LDT descriptor 96 may be similar to the upper half of call gate descriptor 120.

[0062] Turning next to Fig. 10, a block diagram of an instruction format 140 for instructions executed by processor 10 is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 10, instruction format 140 includes a prefix field 142, an opcode field 144, a mod R/M (register/memory) field 146, an SIB (scale index base) field 148, a displacement field 150, and an immediate field 152. Each of the fields except for the opcode field 144 are optional. Thus, instruction format 140 may define a variable length instruction.

[0063] Prefix field 142 is used for any instruction prefixes for the instruction. As described above, an operand size override prefix and an address size override prefix may be encoded into an instruction to override the operating mode of processor 10. These override prefixes are included in prefix field 142. As noted above, the operand size override prefix and address size override prefix may each by bytes included within prefix field 142.

[0064] Opcode field 144 includes the opcode of the instruction (i.e. which instruction in the instruction set is being executed). For some instructions, operands may be specified within opcode field 144. For other instructions, a portion of the opcode may be included within mod R/M field 146. Furthermore, certain opcodes spec-

ify an eight bit or 16 bit register as an operand. Thus opcode encodings may serve to override the defaults indicated by the operating mode of processor 10 as well.

[0065] Mod R/M field 146 and SIB field 148 indicate operands of the instruction. Displacement field 150 includes any displacement information, and immediate field 152 includes an immediate operand.

Computer Systems

[0066] Turning now to Fig. 11, a block diagram of one embodiment of a computer system 200 including processor 10 coupled to a variety of system components through a bus bridge 202 is shown. Other embodiments are possible and contemplated. In the depicted system, a main memory 204 is coupled to bus bridge 202 through a memory bus 206, and a graphics controller 208 is coupled to bus bridge 202 through an AGP bus 210. Finally, a plurality of PCI devices 212A-212B are coupled to bus bridge 202 through a PCI bus 214. A secondary bus bridge 216 may further be provided to accommodate an electrical interface to one or more EISA or ISA devices 218 through an EISA/ISA bus 220. Processor 10 is coupled to bus bridge 202 through a CPU bus 224 and to an optional L2 cache 228. Together, CPU bus 224 and the interface to L2 cache 228 may comprise an external interface to which external interface unit 18 may couple.

[0067] Bus bridge 202 provides an interface between processor 10, main memory 204, graphics controller 208, and devices attached to PCI bus 214. When an operation is received from one of the devices connected to bus bridge 202, bus bridge 202 identifies the target of the operation (e.g. a particular device or, in the case of PCI bus 214, that the target is on PCI bus 214). Bus bridge 202 routes the operation to the targeted device. Bus bridge 202 generally translates an operation from the protocol used by the source device or bus to the protocol used by the target device or bus.

[0068] In addition to providing an interface to an ISA/EISA bus for PCI bus 214, secondary bus bridge 216 may further incorporate additional functionality, as desired. An input/output controller (not shown), either external from or integrated with secondary bus bridge 216, may also be included within computer system 200 to provide operational support for a keyboard and mouse 222 and for various serial and parallel ports, as desired. An external cache unit (not shown) may further be coupled to CPU bus 224 between processor 10 and bus bridge 202 in other embodiments. Alternatively, the external cache may be coupled to bus bridge 202 and cache control logic for the external cache may be integrated into bus bridge 202. L2 cache 228 is further shown in a backside configuration to processor 10. It is noted that L2 cache 228 may be separate from processor 10, integrated into a cartridge (e.g. slot 1 or slot A) with processor 10, or even integrated onto a semiconductor substrate with processor 10.

[0069] Main memory 204 is a memory in which application programs are stored and from which processor 10 primarily executes. A suitable main memory 204 comprises DRAM (Dynamic Random Access Memory). For example, a plurality of banks of SDRAM (Synchronous DRAM) or Rambus DRAM (RDRAM) may be suitable.

[0070] PCI devices 212A-212B are illustrative of a variety of peripheral devices such as, for example, network interface cards, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards. Similarly, ISA device 218 is illustrative of various types of peripheral devices, such as a modem, a sound card, and a variety of data acquisition cards such as GPIB or field bus interface cards.

[0071] Graphics controller 208 is provided to control the rendering of text and images on a display 226. Graphics controller 208 may embody a typical graphics accelerator generally known in the art to render three-dimensional data structures which can be effectively shifted into and from main memory 204. Graphics controller 208 may therefore be a master of AGP bus 210 in that it can request and receive access to a target interface within bus bridge 202 to thereby obtain access to main memory 204. A dedicated graphics bus accommodates rapid retrieval of data from main memory 204. For certain operations, graphics controller 208 may further be configured to generate PCI protocol transactions on AGP bus 210. The AGP interface of bus bridge 202 may thus include functionality to support both AGP protocol transactions as well as PCI protocol target and initiator transactions. Display 226 is any electronic display upon which an image or text can be presented. A suitable display 226 includes a cathode ray tube ("CRT"), a liquid crystal display ("LCD"), etc.

[0072] It is noted that, while the AGP, PCI, and ISA or EISA buses have been used as examples in the above description, any bus architectures may be substituted as desired. It is further noted that computer system 200 may be a multiprocessing computer system including additional processors (e.g. processor 10a shown as an optional component of computer system 200). Processor 10a may be similar to processor 10. More particularly, processor 10a may be an identical copy of processor 10. Processor 10a may be connected to bus bridge 202 via an independent bus (as shown in Fig. 11) or may share CPU bus 224 with processor 10. Furthermore, processor 10a may be coupled to an optional L2 cache 228a similar to L2 cache 228.

[0073] Turning now to Fig. 12, another embodiment of a computer system 300 is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 12, computer system 300 includes several processing nodes 312A, 312B, 312C, and 312D. Each processing node is coupled to a respective memory 314A-314D via a memory controller 316A-316D included within each respective processing node 312A-312D. Addition-

ally, processing nodes 312A-312D include interface logic used to communicate between the processing nodes 312A-312D. For example, processing node 312A includes interface logic 318A for communicating with processing node 312B, interface logic 318B for communicating with processing node 312C, and a third interface logic 318C for communicating with yet another processing node (not shown). Similarly, processing node 312B includes interface logic 318D, 318E, and 318F; processing node 312C includes interface logic 318G, 318H, and 318I; and processing node 312D includes interface logic 318J, 318K, and 318L. Processing node 312D is coupled to communicate with a plurality of input/output devices (e.g. devices 320A-320B in a daisy chain configuration) via interface logic 318L. Other processing nodes may communicate with other I/O devices in a similar fashion.

[0074] Processing nodes 312A-312D implement a packet-based link for inter-processing node communication. In the present embodiment, the link is implemented as sets of unidirectional lines (e.g. lines 324A are used to transmit packets from processing node 312A to processing node 312B and lines 324B are used to transmit packets from processing node 312B to processing node 312A). Other sets of lines 324C-324H are used to transmit packets between other processing nodes as illustrated in Fig. 12. Generally, each set of lines 324 may include one or more data lines, one or more clock lines corresponding to the data lines, and one or more control lines indicating the type of packet being conveyed. The link may be operated in a cache coherent fashion for communication between processing nodes or in a noncoherent fashion for communication between a processing node and an I/O device (or a bus bridge to an I/O bus of conventional construction such as the PCI bus or ISA bus). Furthermore, the link may be operated in a non-coherent fashion using a daisy-chain structure between I/O devices as shown. It is noted that a packet to be transmitted from one processing node to another may pass through one or more intermediate nodes. For example, a packet transmitted by processing node 312A to processing node 312D may pass through either processing node 312B or processing node 312C as shown in Fig. 12. Any suitable routing algorithm may be used. Other embodiments of computer system 300 may include more or fewer processing nodes then the embodiment shown in Fig. 12.

[0075] Generally, the packets may be transmitted as one or more bit times on the lines 324 between nodes. A bit time may be the rising or falling edge of the clock signal on the corresponding clock lines. The packets may include command packets for initiating transactions, probe packets for maintaining cache coherency, and response packets from responding to probes and commands.

[0076] Processing nodes 312A-312D, in addition to a memory controller and interface logic, may include one or more processors. Broadly speaking, a processing node comprises at least one processor and may optionally include a memory controller for communicating with a memory and other logic as desired. More particularly, each processing node 312A-312D may comprise one or more copies of processor 10. External interface unit 18 may includes the interface logic 318 within the node, as well as the memory controller 316.

[0077] Memories 314A-314D may comprise any suitable memory devices. For example, a memory 314A-314D may comprise one or more RAMBUS DRAMs (RDRAMs), synchronous DRAMs (SDRAMs), static RAM, etc. The address space of computer system 300 is divided among memories 314A-314D. Each processing node 312A-312D may include a memory map used to determine which addresses are mapped to which memories 314A-314D, and hence to which processing node 312A-312D a memory request for a particular address should be routed. In one embodiment, the coherency point for an address within computer system 300 is the memory controller 316A-316D coupled to the memory storing bytes corresponding to the address. In other words, the memory controller 316A-316D is responsible for ensuring that each memory access to the corresponding memory 314A-314D occurs in a cache coherent fashion. Memory controllers 316A-316D may comprise control circuitry for interfacing to memories 314A-314D. Additionally, memory controllers 316A-316D may include request queues for queuing memory requests.

[0078] Generally, interface logic 318A-318L may comprise a variety of buffers for receiving packets from the link and for buffering packets to be transmitted upon the link. Computer system 300 may employ any suitable flow control mechanism for transmitting packets. For example, in one embodiment, each interface logic 318 stores a count of the number of each type of buffer within the receiver at the other end of the link to which that interface logic is connected. The interface logic does not transmit a packet unless the receiving interface logic has a free buffer to store the packet. As a receiving buffer is freed by routing a packet onward, the receiving interface logic transmits a message to the sending interface logic to indicate that the buffer has been freed. Such a mechanism may be referred to as a "coupon-based" system.

[0079] I/O devices 320A-320B may be any suitable I/O devices. For example, I/O devices 320A-320B may include network interface cards, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems interface) adapters and telephony cards, modems, sound cards, and a variety of data acquisition cards such as GPIB or field bus interface cards.

[0080] Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

Claims

1. A processor comprising:

   an execution core configured to execute a branch instruction specifying a segment selector,

   wherein said processor is configured to read at least a first entry from a segment descriptor table responsive to said segment selector, and wherein, if said first entry indicates a call gate descriptor, a second entry in said segment descriptor table stores a remaining portion of said call gate descriptor and wherein each of said first entry and said second entry is capable of storing a segment descriptor.

2. The processor as recited in claim 1 wherein said processor is further configured to extract an offset from said call gate descriptor, and wherein at least a first portion of said offset is stored in said first entry, and wherein a remaining portion of said offset is stored in said second entry.

3. The processor as recited in claim 2 wherein said offset is 64 bits.

4. The processor as recited in claim 1 wherein said processor is configured to extract a target segment selector from said call gate descriptor, wherein said target segment selector identifies a target segment descriptor.

5. The processor as recited in claim 4 wherein said target segment descriptor includes a first operating mode indication, and wherein said first operating mode indication establishes a default address size greater than 32 bits.

6. The processor as recited in claim 1 wherein said second entry includes a type field, and wherein said type field is encoded to indicate that said second entry is invalid.

7. The processor as recited in claim 6 wherein said processor, in response to reading said second entry as a segment descriptor, is configured to detect that said second entry is invalid using said type field.

8. A method comprising:

   reading a call gate descriptor from a segment descriptor table, said call gate descriptor comprising a first entry and a second entry in said segment descriptor table, wherein each of said first entry and said second entry is capable of storing a segment descriptor; and

   extracting an offset from said call gate descriptor.

9. A method as recited in claim 8 in which said offset locates a first instruction to be executed in a target code segment.

10. The method as recited in claim 9 wherein said offset is 64 bits.

11. The method as recited in claim 9 further comprising reading a first segment descriptor from only a third entry of said segment descriptor table.

12. The method as recited in claim 9 further comprising extracting a target segment selector from said call gate descriptor, said target segment selector identifying a target segment descriptor.

13. The method as recited in claim 12 further comprising reading said target segment descriptor, said target segment descriptor establishing a default address size greater than 32 bits.

14. The method as recited in claim 9 further wherein each of said first entry and said second entry has a type field.

15. The method as recited in claim 14 further comprising setting said type field in said second entry to indicate invalid.

16. The method as recited in claim 15 further comprising setting said type field in said first entry to indicate a call gate descriptor.

17. The method as recited in claim 15 further comprising:

   attempting to read a segment descriptor from said second entry; and
   determining that said segment descriptor is invalid responsive to said type field in said second entry,

18. A computer system comprising:

   a processor according to claim 1; and an input/output (I/O) device configured to communicate between said computer system and another computer system to which said I/O device is couplable.

19. The computer system as recited in claim 18 wherein said I/O device comprises a modem.

20. The computer system as recited in claim 18 further comprising a second processor comprising:

an execution core configured to execute a branch instruction specifying a segment selector;

wherein said processor is configured to read at least a first entry from a segment descriptor table responsive to said segment selector, and wherein, if said first entry indicates a call gate descriptor, a second entry in said segment descriptor table stores a remaining portion of said call gate descriptor.

Patentansprüche

1. Prozessor mit:

einem Ausführungskern zum Ausführen eines einen Segmentselektor spezifizierenden Verzweigungsbefehls,

wobei der Prozessor in Reaktion auf den Segmentselektor zum Lesen mindestens eines ersten Eintrags aus einer Segmentbeschreibungstabelle vorgesehen ist und, wenn der erste Eintrag einen Aufrufgatterbeschreiber anzeigt, ein zweiter Eintrag in der Segmentbeschreibungstabelle den restlichen Teil des Aufrufgatterbeschreibers speichert und sowohl der erste als auch der zweite Eintrag in der Lage ist, einen Segmentbeschreiber zu speichern.

2. Prozessor nach Anspruch 1, bei dem der Prozessor ferner zum Extrahieren eines Offset aus dem Aufrufgatterbeschreiber vorgesehen ist und mindestens ein erster Teil des Offset in dem ersten Eintrag gespeichert ist und der restliche Teil des Offset in dem zweiten Eintrag gespeichert ist.

3. Prozessor nach Anspruch 2, bei dem das Offset 64 Bits aufweist,

4. Prozessor nach Anspruch 1, bei dem der Prozessor zum Extrahieren eines Target-Segmentselektors aus dem Aufrufgatterbeschreiber vorgesehen ist und der Target-Segmentselektor einen Target-Segmentbeschreiber identifiziert.

5. Prozessor nach Anspruch 4, bei dem der Target-Segmentbeschreiber eine erste Operationsmodusanzeige aufweist und die erste Operationsmodusanzeige eine Standardadressengröße festlegt, die mehr als 32 Bits umfasst.

6. Prozessor nach Anspruch 1, bei dem der zweite Eintrag ein Typenfeld aufweist und das Typenfeld kodiert ist, um anzuzeigen, dass der zweite Eintrag ungültig ist.

7. Prozessor nach Anspruch 6, wobei der Prozessor in Reaktion auf das Lesen des zweiten Eintrags als Segmentbeschreiber unter Verwendung des Typenfelds detektiert, dass der zweite Eintrag ungültig ist.

8. Verfahren mit folgenden Schritten:

Auslesen eines Aufrufgatterbeschreibers aus einer Segmentbeschreibungstabelle, wobei der Aufrufgatterbeschreiber einen ersten Eintrag und einen zweiten Eintrag in der Segmentbeschreibungstabelle aufweist und sowohl der erste Eintrag als auch der zweite Eintrag in der Lage ist, einen Segmentbeschreiber zu speichern; und
Extrahieren eines Offset aus dem Aufrufgatterbeschreiber.

9. Verfahren nach Anspruch 8, bei dem das Offset einen ersten in einem Target-Code-Segment auszuführenden Befehl lokalisiert.

10. Verfahren nach Anspruch 9, bei dem das Offset 64 Bits aufweist.

11. Verfahren nach Anspruch 9, ferner mit dem Schritt des Auslesens eines ersten Segmentbeschreibers nur aus einem dritten Eintrag der Segmentbeschreibungstabelle.

12. Verfahren nach Anspruch 9, ferner mit dem Schritt des Extrahierens eines Target-Segmentselektors aus dem Aufrufgatterbeschreiber, wobei der Target-Segmentselektor einen Target-Segmentbeschreiber identifiziert.

13. Verfahren nach Anspruch 12, ferner mit dem Schritt des Auslesens des Target-Segmentbeschreibers, der eine Standardadressengröße festlegt, die mehr als 32 Bits umfasst.

14. Verfahren nach Anspruch 9, bei dem sowohl der erste Eintrag als auch der zweite Eintrag ein Typenfeld aufweist.

15. Verfahren nach Anspruch 14, ferner mit dem Schritt des Einstellen des Typenfelds in dem zweiten Eintrag derart, dass Ungültig angezeigt wird.

16. Verfahren nach Anspruch 15, ferner mit dem Schritt des Einstellens des Typenfelds in dem ersten Eintrag derart, dass ein Aufrufgatterbeschreiber angezeigt wird.

17. Verfahren nach Anspruch 15, ferner mit folgenden Schritten:

Versuchen, einen Segmentbeschreiber aus dem zweiten Eintrag auszulesen; und
Feststellen, dass der Segmentbeschreiber in Reaktion auf das Typenfeld in dem zweiten Eintrag ungültig ist.

18. Computersystem mit:

einem Prozessor nach Anspruch 1; und
einer Eingangs/Ausgangs- (I/O-) Vorrichtung zur Kommunikation zwischen dem Computersystem und einem weiteren Computersystem, mit dem die I/O-Vorrichtung koppelbar ist.

19. Computersystem nach Anspruch 18, bei dem die I/O-Vorrichtung ein Modem aufweist.

20. Computersystem nach Anspruch 18, ferner mit einem zweiten Prozessor mit:

einem Ausführungskern zum Ausführen eines einen Segmentselektor spezifizierenden Verzweigungsbefehls,

wobei der Prozessor in Reaktion auf den Segmentselektor zum Lesen mindestens eines ersten Eintrags aus einer Segmentbeschreibungstabelle vorgesehen ist und, wenn der erste Eintrag einen Aufrufgatterbeschreiber anzeigt, ein zweiter Eintrag in der Segmentbeschreibungstabelle den restlichen Teil des Aufrufgatterbeschreibers speichert.

Revendications

1. Processeur comportant :

une unité centrale d'exécution configurée pour exécuter une instruction de branchement spécifiant un sélecteur de segment,

dans lequel ledit processeur est configuré pour lire au moins une première entrée à partir d'une table de descripteur de segment en réponse audit sélecteur de segment, et dans lequel, si ladite première entrée indique un descripteur de voie d'appel , une seconde entrée dans ladite table de descripteur de segment stocke une partie restante dudit descripteur de voie d'appel et dans lequel chacune de ladite première entrée et de ladite seconde entrée est capable de stocker un descripteur de segment.

2. Processeur selon la revendication 1 dans lequel ledit processeur est, de plus, configuré pour extraire un prélèvement à partir dudit descripteur de voie d'appel , et dans lequel au moins une première partie dudit prélèvement est stockée dans ladite première entrée et dans lequel une partie restante dudit prélèvement est stockée dans ladite seconde entrée.

3. Processeur selon la revendication 2 dans lequel ledit décalage est de 64 binaires.

4. Processeur selon la revendication 1 dans lequel ledit processeur est configuré pour extraire un sélecteur de segment cible dudit descripteur de voie d'appel , dans lequel ledit sélecteur de segment cible identifie un descripteur de segment de cible.

5. Processeur selon la revendication 4 dans lequel ledit descripteur de segment cible comprend une première indication de mode de fonctionnement et dans lequel ladite première indication de mode de fonctionnement établit une dimension d'adresse par défaut supérieure à 32 binaires.

6. Processeur selon la revendication 1 dans lequel ladite seconde entrée comprend un champ de type et dans lequel ledit champ de type est codé pour indiquer que ladite seconde entrée est invalide.

7. Processeur selon la revendication 6 dans lequel ledit processeur, en réponse à la lecture de ladite seconde entrée en tant que descripteur de segment est configuré pour détecter que ladite seconde entrée est invalide en utilisant ledit champ de type.

8. Procédé consistant à :

lire un descripteur de voie d'appel à partir d'une table de descripteur de segment, ledit descripteur de voie d'appel comprenant une première entrée et une seconde entrée dans ladite table de descripteur de segment, dans lequel chacune de ladite première entrée et de ladite seconde entrée est capable de stocker un descripteur de segment ; et
extraire un prélèvement dudit descripteur de -voie d'appel.

9. Procédé selon la revendication 8 dans lequel ledit décalage place une première instruction à exécuter dans un segment de code cible.

10. Procédé selon la revendication 9 dans lequel ledit décalage est de 64 binaires.

11. Procédé selon la revendication 9 comprenant, de plus, une lecture d'un premier descripteur de segment à partir seulement d'une troisième entrée de ladite table de descripteur de segment.

12. Procédé selon la revendication 9 consistant, de plus, à extraire un sélecteur de segment cible dudit

descripteur de voie d'appel, ledit sélecteur de segment cible identifiant un descripteur de segment cible.

13. Procédé selon la revendication 12 consistant, de plus, à lire ledit descripteur de segment cible, ledit descripteur de segment cible établissant une dimension d'adresse par défaut supérieure à 32 binaires.

14. Procédé selon la revendication 9 dans lequel, de plus, chacune de ladite première entrée et de ladite seconde entrée possède un champ de type.

15. Procédé selon la revendication 14 consistant, de plus, à établir ledit champ de type dans ladite seconde entrée pour indiquer une invalidité.

16. Procédé selon la revendication 15 consistant, de plus, à établir ledit champ de type dans ladite première entrée pour indiquer un descripteur de voie d'appel.

17. Procédé selon la revendication 15 consistant, de plus, à :

essayer de lire un descripteur de segment à partir de ladite seconde entrée ; et
déterminer que ledit descripteur de segment est invalide en réponse audit champ de type de ladite seconde entrée.

18. Système d'ordinateur comprenant :

un processeur selon la revendication 1 ; et un dispositif d'entrée/sortie (I/O) configuré pour communiquer entre ledit système d'ordinateur et un autre système d'ordinateur auquel ledit dispositif d'entrée/sortie peut être couplé.

19. Système d'ordinateur selon la revendication 18 dans lequel ledit dispositif d'entrée/sortie comprend un modem.

20. Système d'ordinateur selon la revendication 18 comportant, de plus, un second processeur comprenant :

une unité centrale d'exécution configurée pour exécuter une instruction de branchement spécifiant un sélecteur de segment ;

dans lequel ledit processeur est configuré pour lire au moins une première entrée à partir d'une table de descripteur de segment en réponse audit sélecteur de segment, et dans lequel, si ladite première entrée indique un descripteur de voie d'appel, une seconde entrée dans ladite table de descripteur de segment stocke une partie restante dudit descripteur de voie d'appel.

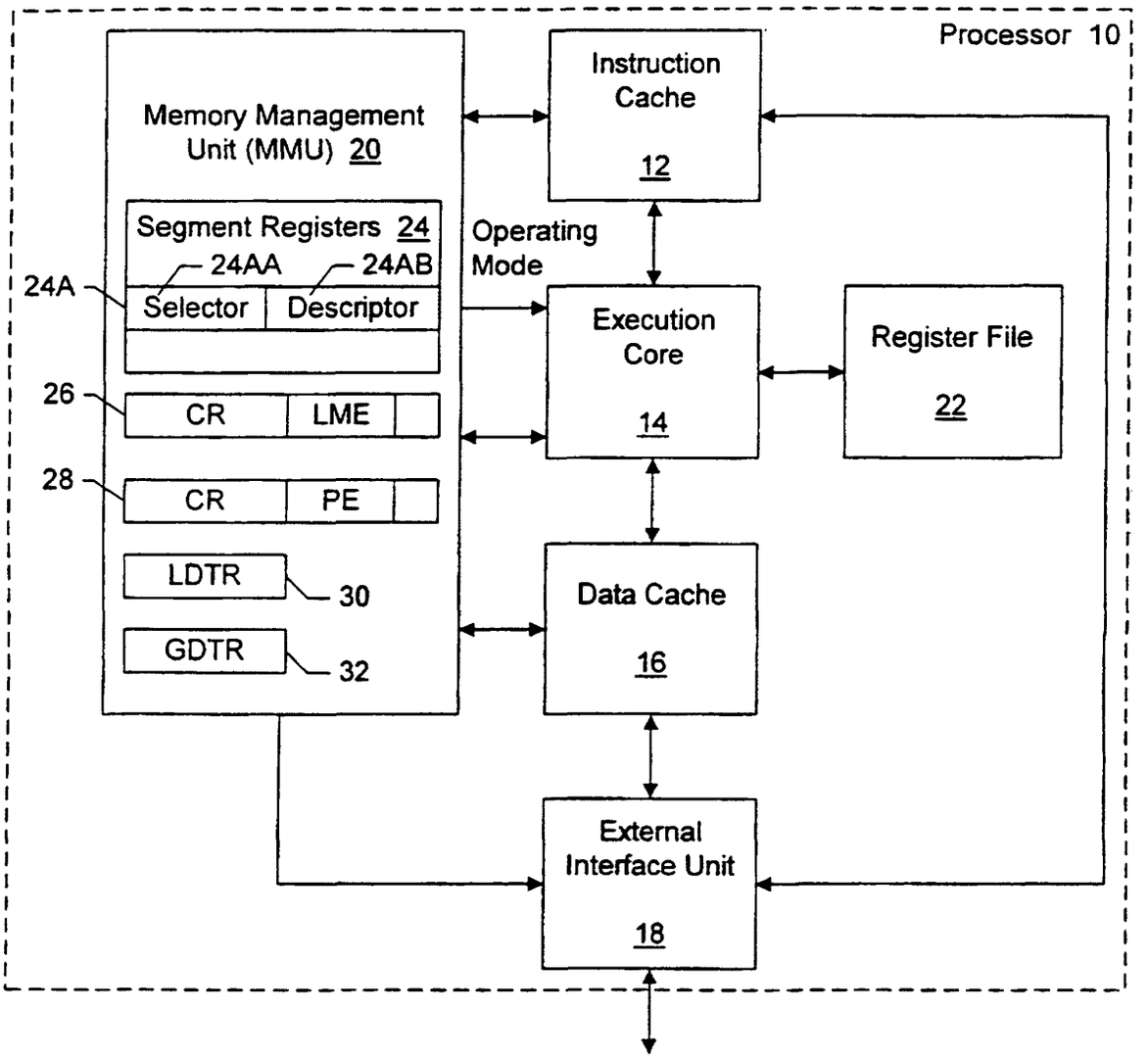
Processor 10
Memory Management Unit (MMU) 20
Segment Registers 24
24AA
24AB
24A
Selector
Descriptor
26
CR
LME
28
CR
PE
LDTR
30
GDTR
32
Instruction Cache
12
Operating Mode
Execution Core
14
Register File
22
Data Cache
16
External Interface Unit
18

Fig. 1

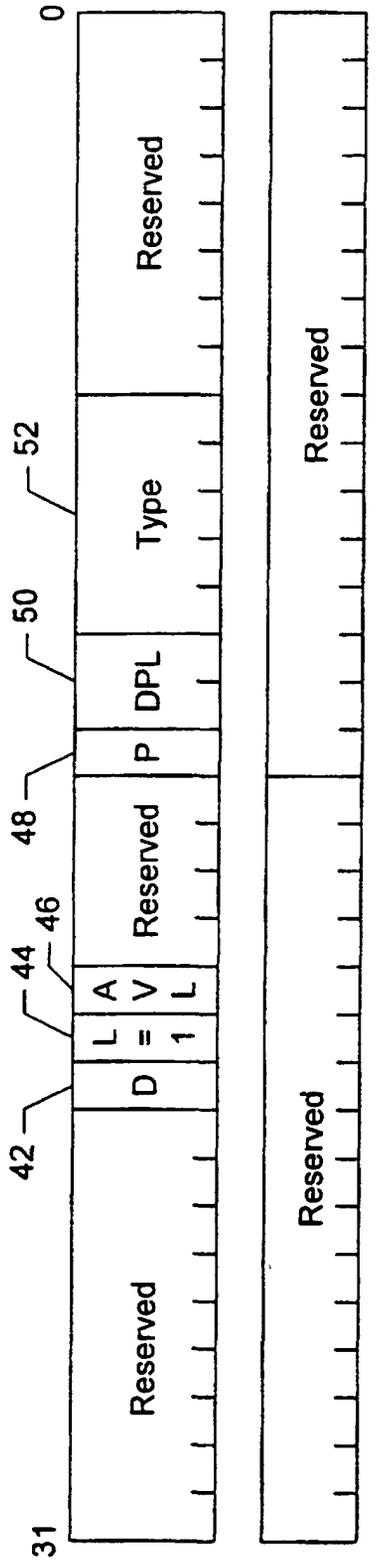
0
Reserved
52
Type
50
DPL
P
48
Reserved
A
V
L
46
44
L
=
1
D
42
Reserved
31
Reserved
40

Fig. 2

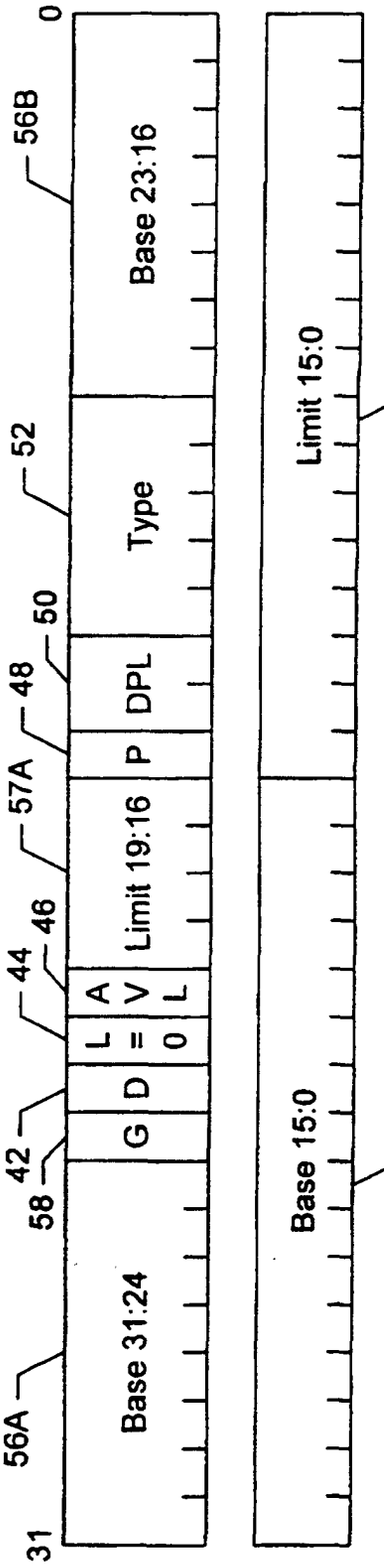
0
56B
Base 23:16
52
Type
50
DPL
48
P
57A
Limit 19:16
A
V
L
46
44
L
=
0
D
G
42
58
Base 31:24
56A
31
Limit 15:0
57B
Base 15:0
56C
54

Fig. 3

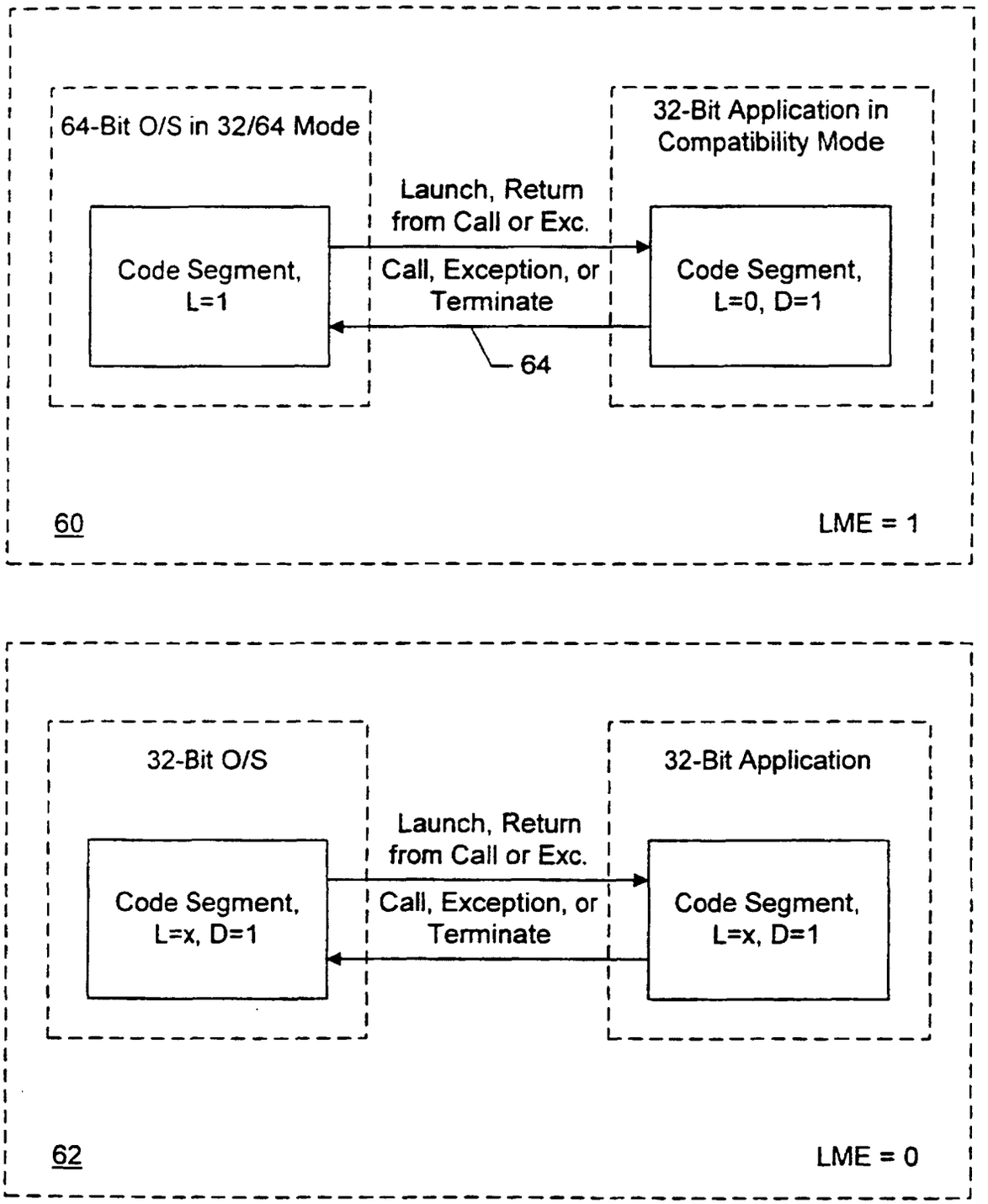
64-Bit O/S in 32/64 Mode
32-Bit Application in Compatibility Mode
Launch, Return from Call or Exc.
Call, Exception, or Terminate
Code Segment, L=1
Code Segment, L=0, D=1
64
60
LME = 1
32-Bit O/S
32-Bit Application
Launch, Return from Call or Exc.
Call, Exception, or Terminate
Code Segment, L=x, D=1
Code Segment, L=x, D=1
62
LME = 0

Fig. 4

LME	CS L Bit	CS D Bit	Operating Mode
0	x	0	16 Bit Mode
0	x	1	32 Bit Mode
1	0	0	16 Bit Compatibility Mode
1	0	1	32 Bit Compatibility Mode
1	1	0	32/64 Mode
1	1	1	Reserved

70

Fig. 5

Number of Override Prefixes (66H Operand, 67H Address)		0	1	2+
32/64 Mode	Operand Size	32 bits	64 bits	16 bits
	Address Size	64 bits	32 bits	32 bits
32 Bit Mode Incl. Compatibility Mode	Operand Size	32 bits	16 bits	16 bits
	Address Size	32 bits	16 bits	16 bits
16 Bit Mode Incl. Compatibility Mode	Operand Size	16 bits	32 bits	32 bits
	Address Size	16 bits	32 bits	32 bits

72

Fig. 6

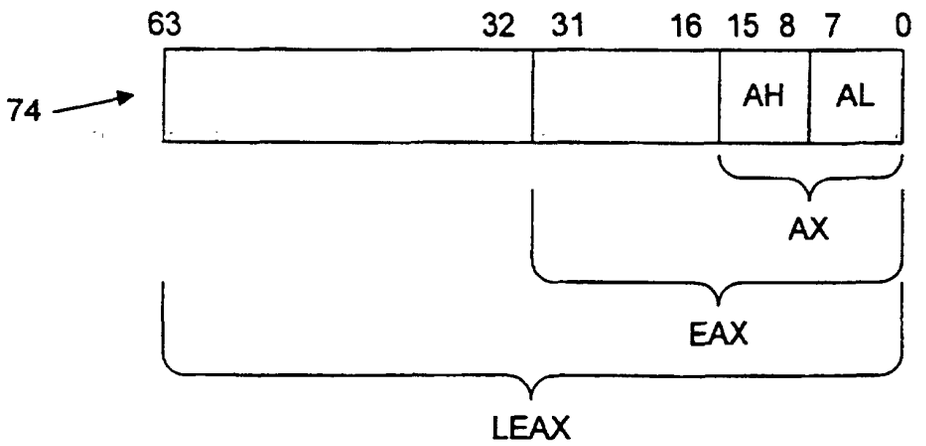

Fig. 7

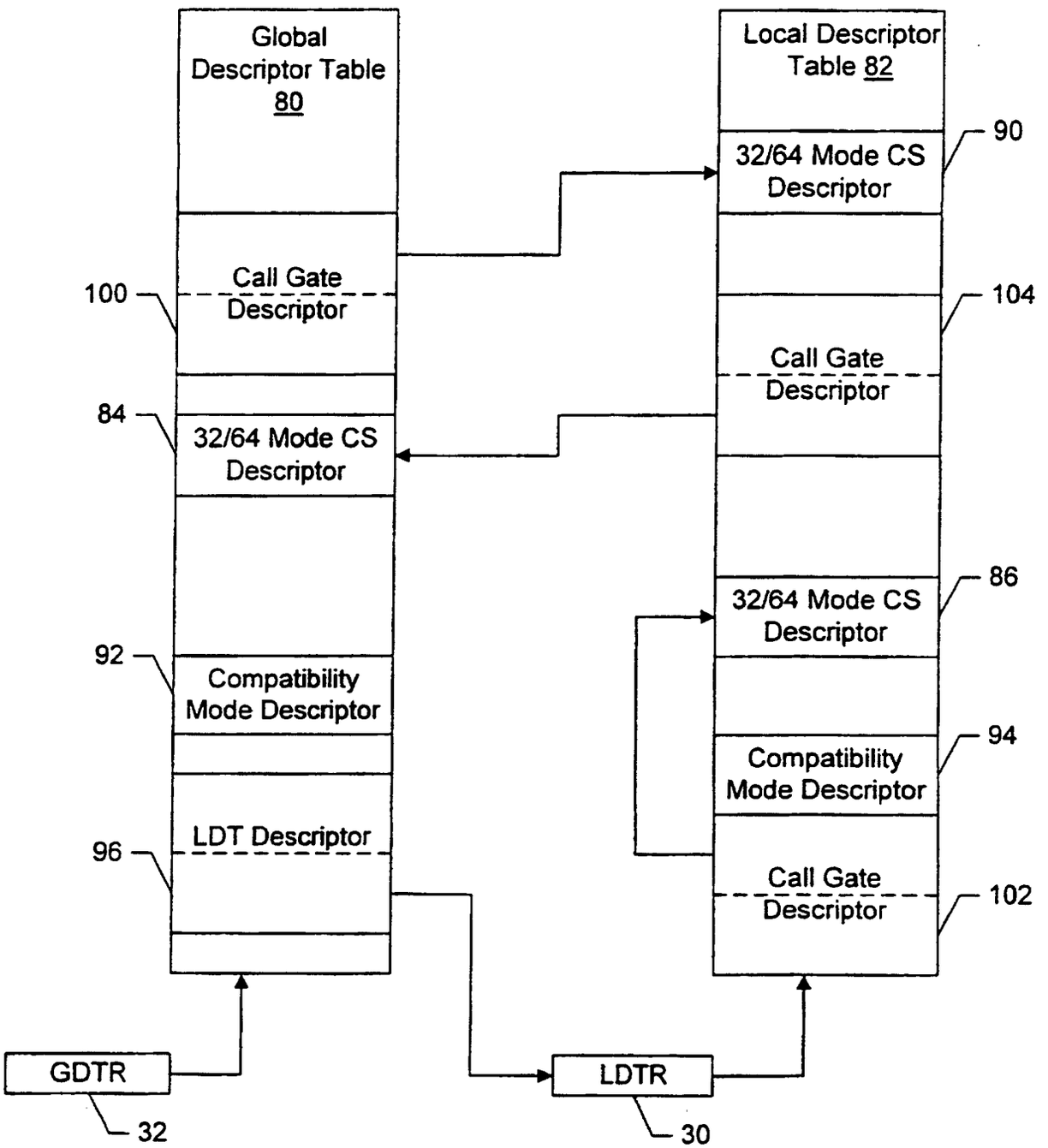
Global Descriptor Table 80
Local Descriptor Table 82
32/64 Mode CS Descriptor
90
Call Gate Descriptor
100
104
Call Gate Descriptor
84
32/64 Mode CS Descriptor
32/64 Mode CS Descriptor
86
92
Compatibility Mode Descriptor
Compatibility Mode Descriptor
94
LDT Descriptor
96
Call Gate Descriptor
102
GDTR
32
LDTR
30

Fig. 8

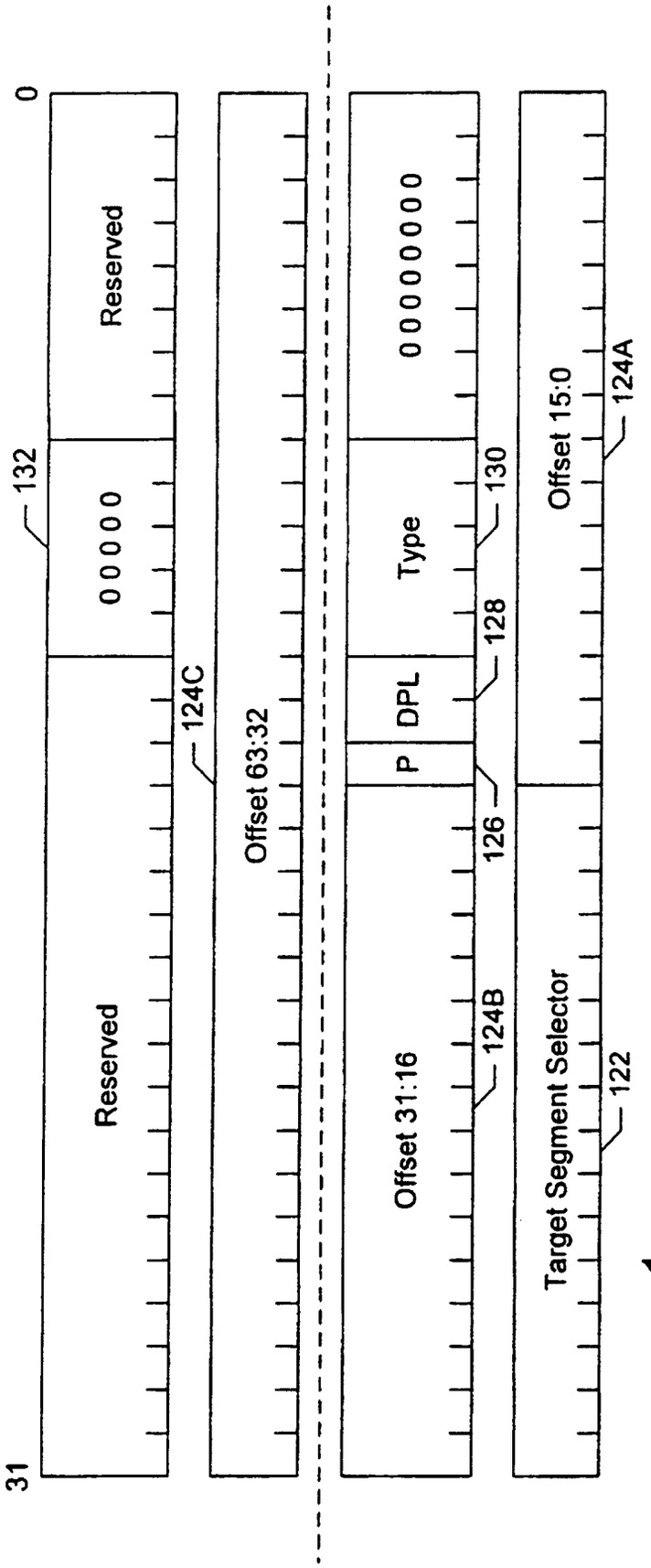
0
31
Reserved
Reserved
132
0 0 0 0 0
124C
Offset 63:32
P
DPL
Type
0 0 0 0 0 0 0 0
128
130
126
Offset 31:16
Offset 15:0
124B
124A
Target Segment Selector
122
120

Fig. 9

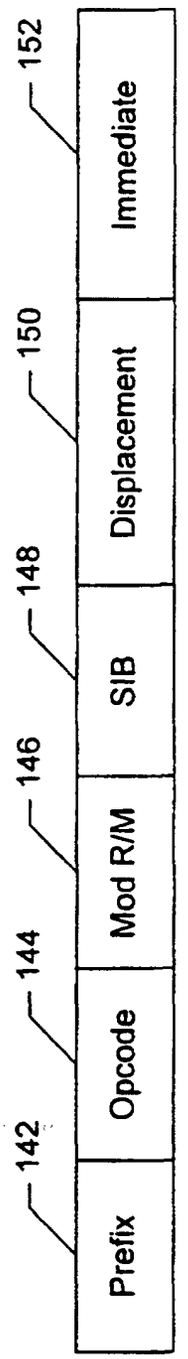
Prefix
Opcode
Mod R/M
SIB
Displacement
Immediate
142
144
146
148
150
152
140

Fig. 10

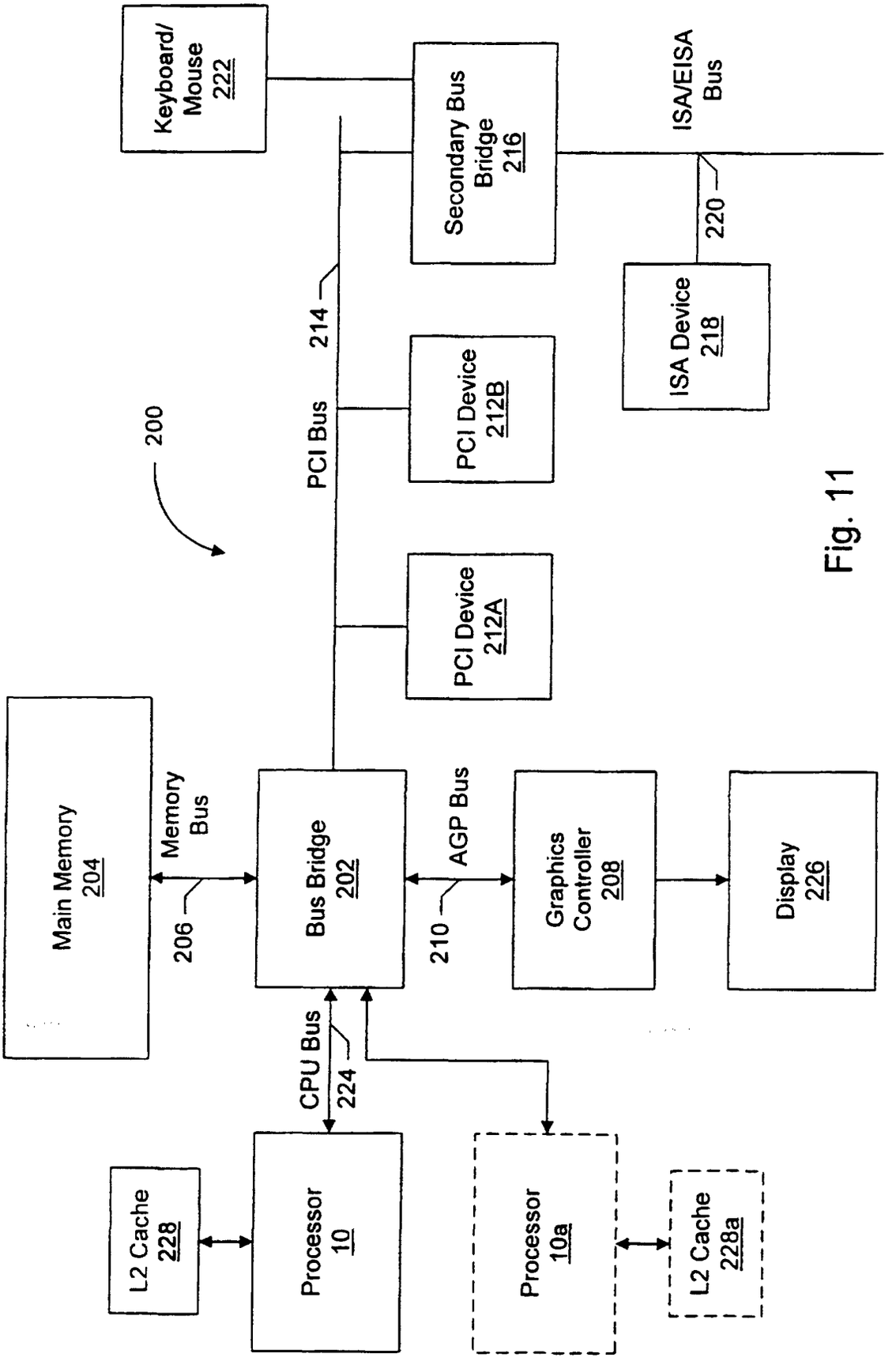
Keyboard/ Mouse 222
Secondary Bus Bridge 216
ISA/EISA Bus
220
ISA Device 218
214
PCI Bus
PCI Device 212B
PCI Device 212A
200
Main Memory 204
Memory Bus
206
Bus Bridge 202
AGP Bus
210
Graphics Controller 208
Display 226
CPU Bus
224
L2 Cache 228
Processor 10
Processor 10a
L2 Cache 228a

Fig. 11

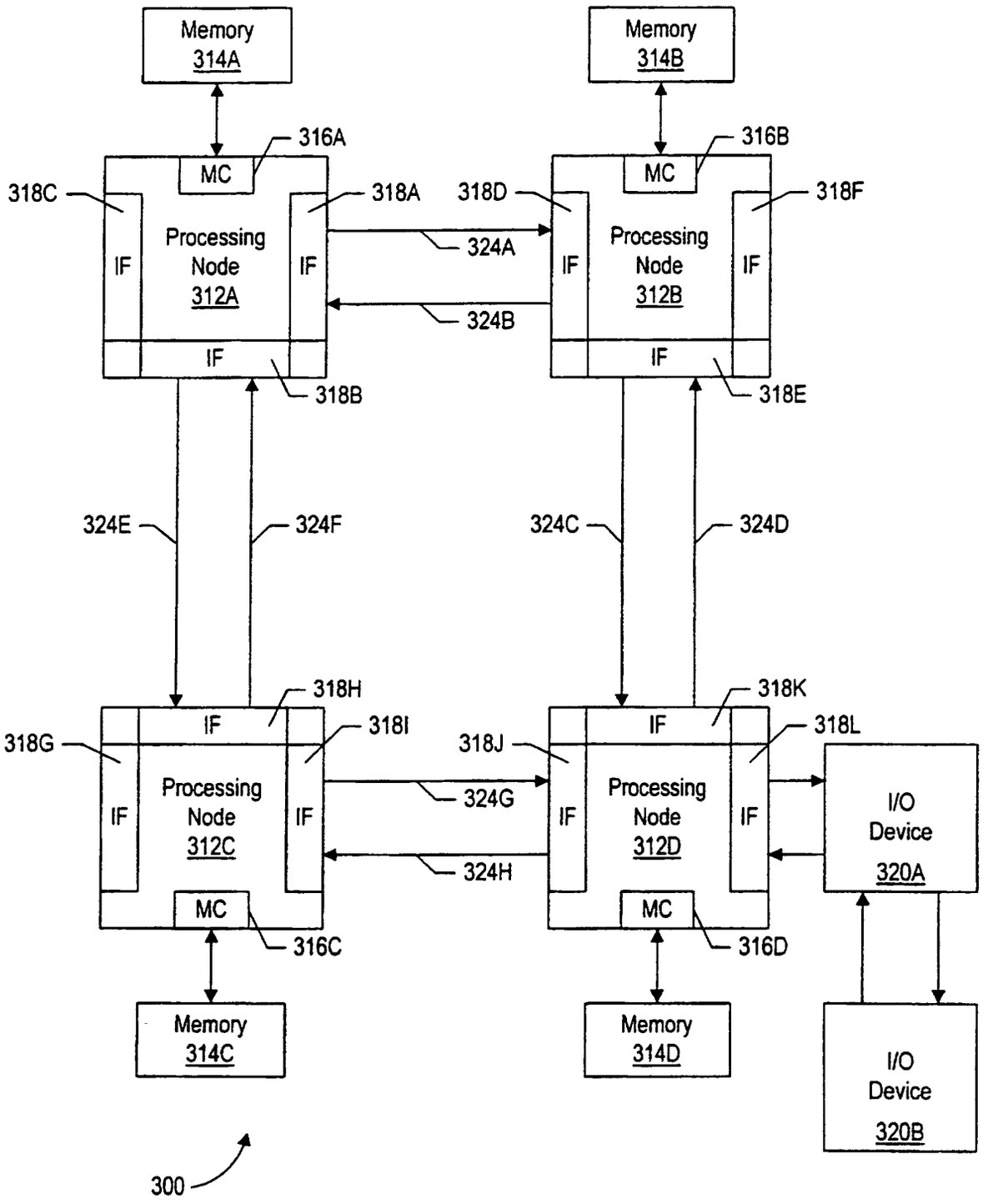
Memory
314A
Memory
314B
316A
316B
MC
MC
318C
318A
318D
318F
IF
Processing
Node
312A
IF
IF
Processing
Node
312B
IF
324A
324B
IF
IF
318B
318E
324E
324F
324C
324D
318H
318K
318G
IF
318I
318J
IF
318L
IF
Processing
Node
312C
IF
324G
IF
Processing
Node
312D
IF
I/O
Device
320A
324H
MC
MC
316C
316D
Memory
314C
Memory
314D
I/O
Device
320B
300

Fig. 12